AutoCTF: Creating Diverse Pwnables via Automated Bug Injection*

Patrick Hulin[1], Andy Davis[1], Rahul Sridhar[2], Andrew Fasano[1], Cody Gallagher[1], Aaron Sedlacek[1], Tim Leek[1], and Brendan Dolan-Gavitt[3]

[1]MIT Lincoln Laboratory
[2]MIT
[3]New York Univeristy
{*patrick.hulin,andrew.davis,andrew.fasano,cody.gallagher,aaron.sedlacek,tleek*}*@ll.mit.edu*
rsridhar@mit.edu
brendandg@nyu.edu

Abstract

Capture the Flag (CTF) is a popular computer security exercise in which teams compete one against the other to attack and/or defend programs in real time. CTFs are currently expensive to build and run: each is a bespoke affair, with challenges and vulnerabilities crafted by experts. This limits both educational value for players and what researchers can learn from them about the human activities such as vulnerability discovery and exploitation. In this work, we take steps towards making CTFs cheap and reusable by extending our LAVA bug injection system to add *exploitable* vulnerabilities, enabling rapid generation of new CTF challenges. New LAVA bug types, including a memory corruption and an address disclosure, form a sufficient set of primitives for program exploitation in most cases. We used these techniques to create AutoCTF, a week-long event involving teams from four universities. For evaluation, we conducted surveys and semi-structured interviews after the event to understand how AutoCTF differed from a handmade CTF, assessing not only challenge realism and difficulty but also the relative effort expended on bug finding and exploit development. Our preliminary results indicate that AutoCTF can form the basis for cost-effective and reusable CTFs, allowing them to be run often and easily to train new generations of security researchers as well as provide empirical data on human vulnerability discovery and exploit development.

1 Introduction

There are over one hundred active CTFs listed on CTF website `ctftime.org`, a testament to the popularity of the activity. Despite the fact that one could apparently play in two a week given this abundance, we would argue this is too few. One problem is variance. No two events are alike, with different flavor, emphasis, and quality. This means it is difficult to use them to train in a focused area. Another issue is that CTF contests are high-profile events and are partly interpreted as a showcase of the organizers' talents. Thus, challenge writers are understandably biased towards creating something totally new—on our past CTF teams, the water-cooler or barroom conversation after an event has indeed mostly focused on the novelty of the puzzles. These forces seem at odds with educational goals, which require repeated practice of core skills.

Nevertheless, CTFs are touted as potentially powerful education and training vehicles [10, 22, 6, 8, 2, 1]. We hypothesize, perhaps controversially, that the top CTFs (DEF CON, Boston Key Party, PlaidCTF, etc.) might not actually be very educationally useful. Rather, they are built to evaluate the relative performance of CTF teams and players. That is, CTFs are baseball games, with DEF CON finals as the World Series. But there isn't currently a clear CTF analogy to Spring Training or even the regular practices of a high school baseball team.

Our aim is to fill this gap with a kind of CTF that is cheap to run and re-run and that is easily configurable with respect to both difficulty and topic. These CTFs will unabashedly reuse the same base applications over and over again to focus attention on vulnerability discovery and exploitation. We make this choice because it allows for reuse, but we note that it can also be justified on realism grounds: practical vulnerability discovery mainly deals in established programs like Firefox and OpenSSL which have been around for years and see frequent updates.

We present AutoCTF, a first step toward reusable, automatically generated CTFs. The basic idea of AutoCTF is to assemble a stockpile of applications into which we can repeatedly inject a handful of exploitable bugs of

*Distribution A: Publc Release. This material is based upon work supported under Air Force Con-tract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opin-ions, findings, conclusions or recommendations expressed in this ma-terial are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force.

known types to create a sequence of fresh CTF competitions. The bugs will be in different places and thus both vulnerability discovery and exploitation will require new effort and even different knowledge and techniques. We use the LAVA bug injection system [11], developed under previous work, extending it to be able to insert *exploitable* bugs of a few key types. These bugs were added to two base programs, one a souped-up `echo` server, and the other a simplified stack based interpreter. This resulted in four auto-generated *challenge* programs which we supplemented with four versions with manually-inserted bugs. AutoCTF ran for a week, during which new challenges were made available each day to teams from four competing universities. The task was to figure out how to exploit the buggy program in order to exfiltrate a flag from a known place on the file system. Four of the eight challenges were solved by the top scoring team.

2 Background

2.1 Capture the Flag

Capture the Flag competitions have been a popular event for over 20 years. In the most general terms, a CTF is a competition in which teams or individuals compete to accomplish some security-relevant goal; upon accomplishing that goal they receive a flag (usually a hard-to-guess string) that can be submitted as proof to the competition organizers.

CTFs are usually divided into two types: *attack-defense*, in which teams run services on a shared network and compete to compromise or disrupt others' services while keeping their own services available, and *jeopardy-style* CTFs, in which teams solve puzzle-like challenges.

The puzzles in jeopardy-style CTFs come in many different flavors; some common types are:

Reverse Engineering Obfuscated programs that must be reverse engineered to reveal a flag.

Pwnables Intentionally vulnerable programs that can be exploited to obtain a flag.

Crypto Weak or poorly implemented cryptography; generally the flag is hidden in an encrypted message that must be decrypted.

Web A web site with some kind of vulnerability (e.g., SQL injection or cross site scripting) that can be exploited to reveal a flag.

These categories are not comprehensive but they provide a sense of the range of challenges that are available. AutoCTF focuses on just one of these categories, pwnables, by injecting exploitable vulnerabilities into a small source program. Since many different vulnerabilities can be added to a given program, this allows many substantially different challenges to be created from the same initial program.

Although CTF challenges are fun, engaging and generally thought to be a good vehicle for cybersecurity education, they are currently very expensive in terms of human time and effort that must be expended. Creating a good pwnable involves many labor-intensive steps: one must write a small program that contains an intentional vulnerability (and, ideally, no other bugs), assess its difficulty, create a sample solution, and test it to make sure the creator's assessment of the challenge difficulty is roughly correct. All of these steps take time and, more importantly, expertise.

To get a sense of the cost (in US dollars) of challenge creation, we examined the contracts awarded by DARPA to create challenges for the Cyber Grand Challenge (CGC). In particular, after consulting with one of the CGC organizers, we focused on the contract [13] awarded to Kaprica Security, Inc, since the other contractors performed additional tasks beyond challenge creation. Since Kaprica was awarded $1.9 million and created 121 challenges for CGC, we can roughly approximate the cost of a challenge at about $15,000. The authors' own estimate for the cost to create an 8-challenge CTF, the 2013 MIT LL CTF [10], is even higher. Challenges for this event and others based on the same infrastructure cost about $25,000 each to design, implement, and test. The reason for this higher cost is that these were attack-defend CTFs, which have more complicated moving parts to get working correctly.

The high cost of challenge creation is usually hidden, as many challenges are created by expert volunteers in their spare time.[1] Unfortunately, this means that organizations with less expertise and resources (or fewer connections) find it difficult to hold CTF competitions. In addition, competitors often produce write-ups of their solutions, so challenges can rarely be reused between events. These factors mean that controlled, focused CTFs cannot currently be run at the scale or frequency that we might wish.

2.2 LAVA

A cheap and plentiful source of bugs in programs is not only useful for CTF competitions; these automatically generated corpora can also be used to evaluate and compare automated bug-finding techniques such as fuzzers, symbolic execution engines, and static analysis tools. In prior work [11], we built a system for Large-scale, Auto-

[1] For example, challenges for the NYU-run CSAW finals are created partly by students and partly by soliciting challenges from experts in the security industry.

```
unsigned int lava_val = 0;
void foo(FILE *f) {
  char x[16];
  fread(x, 16, 1, f);
  // DUA
    lava_val = *(unsigned int *)(&x + 4);
}
...
void bar(char *baz) {
  printf("Value is %s\n",
    // Attack point
    baz + lava_val
      * (lava_val == 0x6176616c));
}
```

Figure 1: An example of a bug inserted by our original LAVA system. Although it is obvious that the pointer baz will go out of bounds when the trigger condition is met, it is highly unlikely the bug will be exploitable.

mated Vulnerability Addition (LAVA). LAVA adds memory corruption bugs to C source code; each bug generated comes with a triggering input that serves as proof that the bug is real. Because AutoCTF builds on LAVA, we will briefly describe the system and its capabilities and limitations here.

LAVA begins with a C source program and an input to that program. In order to add bugs to the program, it finds portions of the input that are currently unused and can be subverted to cause memory corruption errors in the program. This data must be *Dead* (i.e., it does not influence control flow), *Uncomplicated* (not significantly modified from the input), and *Available* somewhere in the program; we call such data a *DUA*. DUAs can then be used to trigger memory safety violations anywhere along the path the program takes on the original input. The site where a DUA is used to trigger a bug is called an *attack point*; in its original form, LAVA attacked pointer arguments to functions by adding the DUA to the pointer value to cause it to go out of bounds. The pointer addition is guarded by a comparison with a trigger value so that the bug only manifests for a single, precisely chosen input. An example of a bug injected by the original LAVA system can be seen in Figure 1.

3 Approach

3.1 Injecting Exploitable Bugs

In the first iteration of the LAVA system, the injected bugs would reliably crash the program (by corrupting pointers) but were not exploitable. Modifying LAVA to be able to inject CTF challenges that are exploitable

```
data_flow[0] = *(unsigned int *)input;
char *ip = ...;
...
if (0x54494246 == data_flow[0]) {
    __asm__(
        "mov %0, %%rsp \n"
        "ret"
         : : "rm" (ip));
}
```

Figure 2: Example of direct stack pointer corruption

```
data_flow[0] = *(unsigned int *)ip;
...
data_flow[1] = *(unsigned int *)input;
...
data_flow[2] = *(unsigned int *)ip2;
...
string[string_pos
        + (0x52544144 == data_flow[0])
            * data_flow[1]]
    = *ip
        + (0x52544144 == data_flow[0])
            * data_flow[2];
```

Figure 3: Example of controlled relative memory write

was a significant effort. We needed to introduce new bug types that would allow memory corruption, and also information leaks to allow bypassing of library ASLR.

For each vulnerability hypothesized by LAVA, we manually analyzed the corresponding source code change to determine if the resulting program was or was not exploitable. To ease this analysis, we stored details about each insertable bug in a SQL database. The LAVA user could then select a particular bug to insert based on any criteria they chose, from source line number distance to observed temporal distance in the original trace.

We added two types of program state corruption bugs: direct stack pointer corruption (Figure 2) and controlled relative memory writes (Figure 3). The first type allows a user of the modified program to pivot the stack pointer to a user-controlled buffer. The second allows them to write user-controlled bytes to a user-controlled offset from an existing pointer in the program, typically on the stack or the heap. Additionally, we added a leak of a variable address enabling ASLR bypass (Figure 4).

Unfortunately, the `clang` tooling infrastructure that LAVA is based on is not built for synthesizing complex additions to the abstract syntax tree, making it difficult to prescribe the injection pattern for a new class of bug. To address this problem, a key element of the modifica-

```
data_flow[0] = *(unsigned int *)keystart;
...
printf("RECALL %lu %s %lu %s\n",
    key->len,
    key->str,
    (0x59465567 == data_flow[0])
        ? &(recall->len)
        : recall->len,
    recall->str);
```

Figure 4: Example of leaking a heap address

tions to LAVA is an embedded domain-specific language (DSL) in our C++ code (Figure 5) allowing rapid construction of new source code changes. This small bit of engineering has made it much easier to stitch together existing source code variables with new code such as the expressions in Figure 3.

3.2 Natural Dataflow

LAVA bugs require access to the DUA at the attack point in order to compare the DUA with a magic value and trigger the bug. Because the DUA may not be in scope at the attack point, LAVA must introduce some form of dataflow to make the DUA or a copy of it available at the time it is needed. As shown in Figure 1, the first iteration of LAVA accomplished this by copying the DUA into a global variable `lava_val` and accessing `lava_val` at the attack point. In order to support multiple bugs in the same program, this approach was extended to use a global array where each entry functioned as a `lava_val` for a different bug. To allow injecting bugs into dynamic shared objects, we added helper methods `lava_set` and `lava_get`, which copy and read the DUA's value to and from the global array.

We were concerned that a hypothetical attacker could identify LAVA bugs by focusing solely on calls to these helper functions and bypass analyzing the rest of the program. To address this issue, we developed an approach that better integrated LAVA's dataflow with the original program: In the program's `main` function, we declare an integer array called `data_flow`. We then modify all non-library function signatures to include an additional first argument of `int* data_flow`, and modify all function calls to include a pointer to the `data_flow` array as a first argument. Finally, as shown in Figure 2, Figure 3, and Figure 4 we modify the injected code at the DUA-site to copy the DUA into the `data_flow` array and modify the injected code at the attack point to reference this array instead of calling `lava_get`. This approach eliminates the need for a global data structure and the `lava_get` and `lava_set` helper functions.

```
LIf(MagicTest(bug).render(), {
    LAsm({ UCharCast(
                LStr(buffer->ast_name)) +
            LDecimal(buffer->selected.low) },
        { "mov %0, %%rsp", "ret" }
    )
})
```

Figure 5: LAVA injection DSL code for bug in Figure 2

3.3 Chaff Bugs

Releasing multiple versions of a program with different injected bugs can easily fall victim to binary comparison tools, which try to identify changes between two versions of a binary. The problem stems from the fact that our injected code is small when compared to the rest of the program's codebase, so comparison quickly yields the injected code. Using differential analysis to discover bugs is a well known technique [12] and several tools have been developed to facilitate this [23]. To prevent this easy win we inject *chaff bugs* into the targets before releasing them.

For this chaff, we use non-exploitable LAVA bugs of the type depicted in Figure 1 injected into random points in the program. Our non-exploitable bugs are still reachable via specially crafted inputs and still crash the program by causing it to dereference unmapped memory. Because these bugs are reachable, an attacker must consider the exploitability of each bug individually. This strategy has the additional benefit of preventing attackers from searching for artifacts left by our system to more quickly find the bugs; after finding the artifacts, the attacker must still distinguish between real bugs and chaff bugs.

4 AutoCTF

We designed a week-long Capture the Flag competition containing eight challenges. Half of these challenges contained vulnerabilities inserted automatically by LAVA and half contained vulnerabilities inserted manually.

Each challenge was hosted in a docker container based on Ubuntu 16.04. We used CTFd [9] as the scoreboard system. Challenges were released at 3pm daily from May 3rd through May 9th as shown in Figure 6 and the competition ended at 3pm on May 10th.

4.1 Base Programs

We developed two simple applications in C that were modified to contain both automatically generated and manually inserted vulnerabilities. We used `xinetd` to connect the services' input and output to the network.

The first of these services, called `blecho`, is a key/value store layered on top of an `echo` server. For each line of text sent to `blecho`, it either stores a value, loads a value and prints it, or ignores it. Values are stored in and retrieved from a temporary directory in the filesystem. This program is 239 lines of source code as measured by David Wheeler's `sloccount`.

The second service, `fifth`, is an interpreter for a binary-format stack-based programming language supporting basic operations: push, add, print, etc. This program is 364 lines long.

4.2 Automated Vulnerability Insertion

Using LAVA, two versions of each service were generated. One had a controlled relative write bug added and the other had a direct stack pointer corruption bug added. These bugs are of the type depicted in Figures 2 and 3, respectively. Both had leaks (à la Figure 4) added to aid in defeating address space layout randomization. To prevent competitors from simply comparing different versions of the services to find vulnerabilities, we also used LAVA to add different non-exploitable bugs to each program (see Section 3.3 for details).

A developer had to create a short configuration file to run LAVA on `blecho` and `fifth`. Once this configuration file was built, LAVA automatically generated a database containing 25937 and 10856 injectable vulnerabilities in `blecho` and `fifth`, respectively. To select a bug to insert, we queried the database to obtain one of the desired exploitable type and instantiated that vulnerability into source using LAVA. We inspected the resulting program to convince ourselves that it should cause the intended effect without additional unintended side effects. Finally, to confirm that a vulnerability was indeed exploitable, we manually created an exploit for all but one of the challenges before releasing them to competitors. The remaining challenge was solved by the winning team.

The time required to add enough exploitable bugs and chaff to a base program as well as vet the output adequately so that it could be used as a challenge was about an hour. Some parts of this process could be speeded up, including database bug selection. However, this hour estimate does not include the time required to verify exploitability, which was more in the several hours range. Speeding up this aspect will be a big focus of future efforts. Note, however, that it is common to both automated and manual bug insertion.

4.3 Manual Vulnerability Insertion

We released four challenges containing vulnerabilities that were manually inserted into our two base programs, `blecho` and `fifth`.

The first vulnerability inserted into `blecho` removed logic that prevented keys from containing non-alphanumeric characters. Since the key was used as a filename, removing this logic allowed competitors to use a path traversal attack to read a flag. The second vulnerability added to `blecho` was a controlled relative write, similar to one of the LAVA generated vulnerabilities, but with a much smaller range of possible addresses to write to. This vulnerability allowed corruption of `blecho`'s storage directory, which in turn allowed arbitrary file reads.

The two vulnerabilities inserted into `fifth` added a stack overflow and a use-after-free. Exploitation of either allowed full control of execution.

The time required to manually insert a bug varied from a few tens of minutes to several hours. This variance is interesting and is explained by three factors. First, if the original author of the base program was adding a bug, this took much less time since that individual already understood program function, control-flow, and data structures deeply. Second, if the insertion was very early in the program execution this was easiest since almost none of the progam needed to be understood. Third, if the bug type was to be subtle, involving limited but adequate control of a pointer that could overwrite sensitive program data, then that sort of insertion necessarily required deep knowledge of the program and took the longest to get right. By contrast, auto-injected bugs using LAVA, while not always subtle, were always of the order of an hour to create.

5 Results

5.1 Event

We ran AutoCTF over the course of a week for four university security clubs in May of 2017. Three teams solved at least one challenge, with the winning team solving four of the eight challenges. Four teams were initially registered, from four separate universities, but one team dropped out and, generally, participation was low due to conflicts with final exams and projects. For future iterations of AutoCTF, we will choose a better time slot.

Date	Challenge	Created by	Vulnerability type
May 3	`blecho` day 1	LAVA	Controlled relative write (Figure 3)
May 4	`blecho` day 2	LAVA	Direct stack pointer corruption (Figure 2)
May 6	`blecho` day 4	Human	Path traversal
May 8	`blecho` day 6	Human	Limited controlled relative write
May 3	`fifth` day 1	Human	Stack overflow
May 5	`fifth` day 3	Human	Use-after-free
May 7	`fifth` day 5	LAVA	Controlled relative write (Figure 3)
May 9	`fifth` day 7	LAVA	Direct stack pointer corruption (Figure 2)

Figure 6: Schedule and description of each vulnerability

5.2 Interviews

We interviewed six of the players to get their feedback on the event. The main conclusion we drew was that the challenges were probably too difficult for a small event, as some of the participants were fairly new to the CTF world.

The participants had somewhat conflicting opinions on the reuse of base programs. Some said they enjoyed the repetition, as it meant they could build on their reverse engineering experience from previous iterations of each program. Unfortunately, repetition inherently leads to less variety in challenges, which a few participants disliked. One said that, while the reverse engineering of each challenge iteration was easier and faster, they did not enjoy the task of transcribing notes from one IDA Pro database to another. All said they would play if the event were held again with different bugs in the same base programs. It is worth noting that the team with the most solves seemed to prefer one of the base programs (`fifth`), which constituted three of their four solves. In interviews, it became clear that they had invested a fair amount of RE in that program and had insufficient resources to spend as heavily on `blecho`. This seems an interesting aspect to investigate in future AutoCTFs.

Participants were also split on whether it was more difficult to find the bugs or to exploit the program. One player we interviewed, who had significant experience playing CTFs (he had played in more than 10 events), thought that due to the base program reuse the difficulty was almost entirely in exploitation, and even suggested it as a way to train exploit development skills independently from reverse engineering skills. On the other hand, at least one player was fairly stymied by the chaff—although he found the injected chaff "fairly transparent," he was not able to determine which were exploitable, and commented that when there are many crashes but few are exploitable it can be "demotivating."

One repeated negative comment was that, especially to a human reverse engineer, the LAVA magic value comparisons strongly stand out (Figure 3, line 8). Further, few teams seemed particularly slowed by chaff bugs; in interviews they indicated that they were fairly transparently not exploitable. These LAVA injection artifacts and deficiencies will be an important area for future work.

5.3 Discussion

These challenges were vastly easier and cheaper (in terms of time) to create than challenges that some of the authors had made in past CTFs, although verifying exploitability still took significant time. While LAVA assists in this effort by generating an input that triggers the bug, transforming such a crashing input into a working exploit is always an involved task. Further reducing exploitability verification time should be possible and is an ongoing effort. This is an area where CTF exploitation diverges from the real world, as we think that the ratio between time spent on vulnerability discovery and exploitation tends more towards discovery in real-world programs, which are usually much larger than a few hundred lines.

We believe that in the future, given a small stable of base programs, that we could easily run another AutoCTF event with little effort.

In terms of player experience, we found that the challenges were probably more difficult than we had anticipated. For example, at NYU, we saw 15-20 students participate on the first day; however, almost all of these students were very new to CTFs and security in general, and all but two (who were the most experienced at playing CTFs) dropped out after that first day. One interesting nuance here is that much of the difficulty was in the exploitation phase. The factors that influence how challenging a bug is to exploit include things like the size of the binary (and hence the availability of ROP gadgets) and what exploit mitigations are turned on. These are not under the direct control of LAVA right now, and so for future CTFs we may need to extend the system further to exert control over these features and thereby tune the difficulty more precisely.

6 Limitations

AutoCTF has three main limitations: vulnerability types, LAVA-required application source editing, and exploitation difficulty.

The LAVA system, when we began this work, was able to inject memory corruption bugs. Thus, it is hardly surprising that the class of CTF challenges automatically created for AutoCTF involves out-of-bounds reads and writes on the stack and heap. As discussed in Section 3, controlled pointer writes and `printf`-based read disclosures were identified as fairly straightforward LAVA bug type extensions that would nevertheless provide sufficient offensive power for AutoCTF players to practice modern exploitation techniques. However, this means other kinds of exploitable bugs, both simple and complicated, are not presently within its repertoire. For instance, LAVA cannot inject any of the following for AutoCTF: directory traversal, use-after free, and more general read disclosures. These bug types all seem possible with LAVA, and we have begun to think concretely about how we might implement them. Other bug types such as side channels, crypto and logic flaws seem more fundamentally out of reach. Our intuition is that this is actually because they are rather broad and ill defined; implementing an exploitable bug type in LAVA requires a precise formulation. It is possible that there are specific classes of bugs within these seemingly trickier categories that could be part of AutoCTF's future, once clearly specified.

LAVA bugs, additionally, are not always injectable in a freshly written challenge program. One cause for this is a mismatch between the syntactic constructs recognized by LAVA in terms of Clang's AST matchers and those found in the source as written by a programmer. For instance, LAVA identifies attack points for injecting exploitable bugs as follows.

1. A memory access attack point is an assignment in which the left-hand-side is
   (a) a pointer dereference such as `*p='\0'`, or
   (b) an index into an array such as `a[i]=7`
2. A `printf` attack point is a `printf` call containing an integer argument, e.g. `printf("%d\n", x)`

These are the only places in a program source where LAVA can inject code such that a memory corruption or information leak can manifest there. If none of these constructs appear in a program, LAVA will not find any locations where it can attempt to add a bug. If a program doesn't use arrays or pointers, LAVA can't add bugs to it. This means code may have to be partly re-written in terms of these constructs for it to be usable in AutoCTF.

Additionally, LAVA will be unable to inject bugs into a program that uses data in such a sparing way that there are no or too few DUAs. This can be assessed by running LAVA noting how many DUAs it locates, and then modifying the program to increase that number. LAVA locates DUAs as tainted (attacker-controlled) data at particular points in a program trace that satisfy the following requirements. A DUA is

1. at least as big as a machine pointer
2. not used to decide many previous branches
3. not a complicated function of input bytes

One can increase the number of DUAs available in a program only with a detailed understanding of the current data flow. For instance, if one knows where data is first read in, one might introduce additional buffers in which that data is needlessly stored, and LAVA will find and use them to create bugs. Note that the number of potential bugs injectable by LAVA is roughly linear in the product of the number of DUAs and the number of attack points, so we want to make both numbers large.

The bugs injected by LAVA also currently have an easily recognizable trigger—a four-byte "magic value"; as seen in Section 5, several different players noticed this feature and found it unrealistic. Although our chaff injection prevented this from being used as a shortcut to figure out where the exploitable bug was injected, chaff can also be frustrating to players. We hope to create more natural triggers in the future by splitting up the comparison into multiple smaller comparisons, applying transformations to the DUA before the trigger comparison, and integrating it more tightly with the program's existing state and data structures.

A final limitation of AutoCTF is that, given a LAVA injected bug, exploitability must be verified manually. This puts a lower bound on the time to auto-generate a challenge that is higher than we would like. That is, creating a new challenge with LAVA might take fifteen minutes, but proving that it is exploitable may take several hours. This situation sees a parallel in AutoCTF game play, where players observed about an order of magnitude difference between the time to find the bug and the time to exploit it. Further, note that players usually chose to exploit LAVA bugs via standard ROP techniques which entails a fairly lengthy but bounded exploit development process. We never observed a player choosing to exploit LAVA bugs to corrupt application-specific data such as length fields and directory string contents. This indicates a bias in players to choose exploit strategies with known time requirements over investing in understanding and exploring a binary to find subtle data attacks that might be much easier to stage. This is interesting and

we will would like to frame experiments to measure this bias in the future. More importantly, we believe, for AutoCTF to be viable, we will need to invest in analyses that speed exploit development. These analyses would leverage the advantage that we know the exact input required to control a LAVA-injected bug and precisely where it manifests.

7 Future Work

In the future, we would like to explore just how much of a CTF can be automated, in order to put CTFs within the reach of as many people as possible. At the same time, we would like to improve the quality of the automatically generated challenges and ensure sufficient diversity. We discuss several areas of study needed to achieve that goal here, and consider what research might be enabled by such a system.

7.1 Bug Diversity and Realism

The bugs we inject currently require very few prerequisites for exploitation. This increases the odds that a given bug will be exploitable, but limits the types of bugs we can inject. As discussed in the previous section, we believe that many types of memory safety bugs (both spatial and temporal) are within reach; other bugs, such as timing channels and cryptographic weaknesses, will require more fundamental research in order to precisely specify and add to an existing program. This research would benefit not only automatically generated CTFs, but also our ongoing attempts to automatically create high-quality vulnerability corpora for evaluating bug discovery tools.

LAVA currently produces an input to trigger each bug it injects, but not an exploit for the bug. Automatically generating exploits is a studied academic problem [5] but remains a complex, open-ended task in most cases. With LAVA we have a simpler problem since we control the bug we are injecting and can modify the source or binary code of the program. Given our advantages, LAVA should be able to provide more assistance to the person tasked with proving the exploitability of the bug or even provide the proof (in the form of a working exploit) itself.

7.2 Improving Automation

Aside from proving exploitability, we still require human intervention to create the base programs and to assign difficulty scores. We believe that even these steps might be automated, however.

Creating small challenges by hand is not insurmountably difficult, but it does pose some risks: *unintended* exploitable bugs introduced in a base program will be present in every challenge derived from it, which could allow a large number of problems to be solved in the same way. Instead of crafting the base programs by hand, we could trawl GitHub to look for small C programs that read from `stdin` and write to `stdout`. Most such programs will be too large to be reasonably reverse engineered during a CTF, but we may be able to use techniques such as program slicing [21] to make the programs a more manageable size. The binary comparison problem described in Section 3 arises in a different form here: since the base programs are widely available, participants might be able to obtain them and compare them with our buggy version to locate the added bug. Our chaff technique should work here as well, but we also plan to investigate techniques such as binary stirring [20] to help prevent comparison.

A thornier challenge is difficulty estimation. As exemplified in this CTF, where we inadvertently made the challenges too hard for novice players, even human judgement is not always very accurate in estimating the difficulty of a challenge. Difficulty is also influenced not only by source-level features of the bug injected, but by features of the binary program (e.g., the availability of ROP gadgets) and the runtime environment (e.g., exploit mitigations such as ASLR). By running more and larger-scale automatically generated CTF competitions, we hope to identify features of programs, bugs, and environments that contribute to the difficulty of an exploitation challenge and use those features to automatically assign point values to the generated challenges.

7.3 Researching Human Vulnerability Discovery

Although automated tools have made great strides in recent years [3], humans still hold an advantage when it comes to finding deep, subtle vulnerabilities. However, just *how* humans go about finding security vulnerabilities is not well understood, in part because it is hard to carry out large-scale controlled experiments. CTFs provide an opportunity to study security practice in a controlled environment. In the future, we would like to use a record-replay system to record the actions of players for later study. Such data collection would allow us to understand how strong players do their work, examine weaker players' behavior in order to help them improve, and understand more about the underlying vulnerabilities and how players find and exploit them. We believe this research has the potential to not only improve cybersecurity education but also generate insights that can be applied to improving automated bug-finding tools as well.

8 Related Work

Automatic problem generation for educational assessment is an active research area [14, 18, 17, 15], especially with the rise of online education and computer based learning. Most prior work focuses on more traditional educational environments, but some work has been done in applying these ideas to computer security and CTFs in particular.

We are not the first to recognize the value of automatically generating CTF challenges. We are, however, the first to be able to inject bugs into existing programs instead of using substitution-based approaches or domain specific languages. Previous approaches give the challenge author more control over the generated challenge in exchange for limiting the diversity of the challenges generated per template.

The first work in automatic challenge generation was done by Burket et al [7]. Their event, picoCTF [4], is targeted at middle and high school students, so the difficulty must be low. Because of this, they focus on challenges in categories other than memory corruption attacks. For example, they might automatically change the key of a simple cipher on a per-team basis.

Their competition also has a cash prize, and thus cheating is a real threat. Therefore, their work focuses on catching teams who are sharing answers. Their approach is to template the challenges and then use a system to automatically fill out the templates on a per team basis with a unique flag and other per team parameters. Another artifact of the cash prize is that they need to ensure a consistent difficulty between the generated challenges.

Building on the work by Burket et al, Gábor Szarka developed Blinker [19], a domain specific language to describe challenges. The majority of the changes from the previous work is a focus on binary challenges using a custom LLVM toolchain. Another tool Blinker provides is an automation framework for creating network forensics challenges. Currently, the author is running an online capture the flag event using his framework. There have been no published results for this event as of the writing of this paper.

Pewny and Holz created a similar system to LAVA called EvilCoder [16], which subverts attacker-controlled data to remove security checks in source code. Because EvilCoder uses a static approach, it does not have the ability to easily generate triggering inputs to prove the existence of its bugs. LAVA bugs come with a triggering input and thus give the LAVA user a head start in demonstrating their exploitability.

9 Conclusion

This paper introduced AutoCTF, a jeopardy-style computer security competition employing automatically generated vulnerabilities. These synthetic bugs, injected using an extended version of the LAVA system, varied in type, including controlled relative writes, read disclosures, and stack pointer corruption abilities. Together, these provided sufficient offensive power for exploitation. Teams playing AutoCTF solved challenges involving both LAVA and manually injected bugs during the competition, indicating a rough equivalence. AutoCTF achieved considerable code reuse, with four buggy versions each of only two base programs. Half of the CTF was completely auto-generated, making that portion very inexpensive. Our experience suggests that, with some work to reduce artifacts, and a better-set difficulty level, AutoCTF might be run next time using only LAVA bugs, dramatically reducing cost. In the future, we imagine AutoCTF might be set up to run virtually without human intervention and provide an inexhaustible training ground for those wanting to practice vulnerability discovery and exploit development.

10 Acknowledgments

Many thanks to all the players in our CTF, with special thanks to Nick Gregory, Josh Hofing, Will Blair, Nick Burnett, and Toshi Piazza, who agreed to be interviewed for this work.

References

[1] Capture the flag — ctf field guide. `https://trailofbits.github.io/ctf/ctf.html`.

[2] CSAW: Cyber security awareness week. `https://csaw.engineering.nyu.edu/ctf`.

[3] DARPA. Cyber Grand Challenge. `http://archive.darpa.mil/cybergrandchallenge/`.

[4] picoCTF. `https://picoctf.com/`, 2017. Accessed 16 May 2017.

[5] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., AND BRUMLEY, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[6] BASHIR, M., LAMBERT, A., GUO, B., MEMON, N., AND HALEVI, T. Cybersecurity competitions: The human angle. *IEEE Security & Privacy 13*, 5 (2015), 74–79.

[7] BURKET, J., CHAPMAN, P., BECKER, T., GANAS, C., AND BRUMLEY, D. Automatic problem generation for capture-the-flag competitions. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)* (2015), USENIX Association.

[8] CHEUNG, R. S., COHEN, J. P., LO, H. Z., ELIA, F., AND CARRILLO-MARQUEZ, V. Effectiveness of cybersecurity competitions. In *Proceedings of the International Conference on Security and Management (SAM)* (2012), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 1.

[9] CHUNG, K. CTFd a capture the flag framework. https://ctfd.io, 2017. Accessed 16 May 2017.

[10] DAVIS, A., LEEK, T., ZHIVICH, M., GWINNUP, K., AND LEONARD, W. The fun and future of ctf. In *3GSE* (2014).

[11] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. LAVA: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 110–121.

[12] DULLIEN, T., AND ROLLES, R. Graph-based comparison of executable objects (english version). *SSTIC 5* (2005), 1–3.

[13] FEDERAL PROCUREMENT DATA SYSTEM NEXT GENERATION. Contract FA875014C0189. https://www.fpds.gov/ezsearch/fpdsportal?indexName=awardfull&templateName=1.4.2&s=FPDS&q=PIID:"FA875014C0189", 2015. Accessed 16 May 2017.

[14] GULWANI, S., ANDERSEN, E., AND POPOVIĆ, Z. A trace-based framework for analyzing and synthesizing educational progressions.

[15] MEYER, J. P., AND ZHU, S. Fair and equitable measurement of student learning in moocs: An introduction to item response theory, scale linking, and score equating. *Research & Practice in Assessment 8* (2013), 26–39.

[16] PEWNY, J., AND HOLZ, T. EvilCoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), ACM, pp. 214–225.

[17] SADIGH, D., SESHIA, S. A., AND GUPTA, M. Automating exercise generation: A step towards meeting the mooc challenge for embedded systems. In *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education* (New York, NY, USA, 2013), WESE '12, ACM, pp. 2:1–2:8.

[18] SINGH, R., GULWANI, S., AND RAJAMANI, S. Automatically generating algebra problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence* (2012), AAAI'12, AAAI Press, pp. 1620–1627.

[19] SZARKA, G. Blinker. https://gs509.user.srcf.net/blinker/, 2016. Accessed 16 May 2017.

[20] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12.

[21] WEISER, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (Piscataway, NJ, USA, 1981), ICSE '81.

[22] WERTHER, J., ZHIVICH, M., LEEK, T., AND ZELDOVICH, N. Experiences in cyber security education: The MIT Lincoln Laboratory capture-the-flag exercise. In *CSET* (2011).

[23] ZYNAMICS. Bindiff. https://www.zynamics.com/bindiff.html, 2011. Accessed 16 May 2017.